NAME: ABHIMANYU SINGH

SECTION: K18LE

REGISTRATION NO. : 11800740

ROLL NO. : 07

GITHUB: - https://github.com/luciferdevill/OSCA3/tree/master

Problem 20: There are 3 student processes and 1 teacher process. Students are supposed to do their assignments and they need 3 things for that pen, paper and question paper. The teacher has an infinite supply of all the three things. One students has pen, another has paper and another has question paper. The teacher places two things on a shared table and the student having the third complementary thing makes the assignment and tells the teacher on completion. The teacher then places another two things out of the three and again the student having the third thing makes the assignment and tells the teacher on completion. This cycle continues. WAP to synchronize the teacher and the students.

GitHub Link: https://github.com/luciferdevill/OSCA3/blob/master/q20.c

1. Description:-

Assume a student requires three things to make the assignment: pen, paper, and question paper. There are three students around a table, each of whom has an infinite supply of one of the three ingredients — one student has pen, another has paper, and the third has question paper.

There is also a teacher who gives the student to make their assignment by arbitrarily (non-deterministically) selecting two of the supplies to place on the table. The student who has the third supply should remove the two items from the table, using them (along with their own supply) to make an assignment, which they give for a while. Once the student has finished his assignment, the agent places two new random items on the table. This process continues forever.

Three semaphores are used to represent the items on the table; the teacher increases the appropriate semaphore to signal that an item has been placed on the table, and students decrement the semaphore when removing items. Also, each student has an associated semaphore that they use to signal to the teacher that they are done making assignment; the

teacher has a process that waits on each student's semaphore to let it know that it can place the new items on the table.

This seems like a fairly easy solution. The three student processes will make an assignment and complete it. If they can't make an assignment, then they will go to sleep. The teacher process will place two items on the table, and wake up the appropriate student, and then go to sleep. All semaphores except lock are initialized to 0. Lock is initialized to 1, and is a mutex variable. The student immediately sleeps. When the teacher puts the two items on the table, then the teacher will wake up the appropriate student. The student will then grab the items, and wake the teacher. While the student is making the assignment, the teacher can place two items on the table, and wake a different student (if the items placed aren't the same). The teacher sleeps immediately after placing the items out.

2. Algorithm:

1. The main function gets to be executed and under it particular function should be executed given with the threads consecutively.
2. The threads are created and search for the consecutively for the values.
3. With the first thread the first function is gets executed and over there the teacher gives resources to the students.
4. With the second thread the function is get executed i.e. studenti will take all the resources and complete its process.
5. With the third thread the function is get executed i.e. studenti will take all the resources and complete its process.
6. With the fourth thread the function is get executed i.e. studenti will take all the resources and complete its process.
7. The Output will be printed in the order of the threads and function is used loop will run upon the which last student completed the assignment.

3. Complexity: - O(logn)

4. Constraints:-

Programming using Synchronisation makes life harder as utmost care must be taken to ensure Ps and Vs are inserted correspondingly and in the correct order so that mutual exclusion and deadlocks are prevented. In addition, it is difficult to produce a structured

layout for a program as the Ps and Vs are scattered all over the place. So the modularity is lost. Semaphores are quite impractical when it comes to large scale use. Semaphores involve a queue in its implementation. For a FIFO queue, there is a high probability for a priority inversion to take place wherein a high priority process which came a bit later might just have to wait when a low priority one is in the critical section. For example, consider a case when a new student joins and is desperate to do assignment. What if the teacher who handles the distribution of the resources follows a FIFO queue (wherein the desperate student is last according to FIFO) and chooses the resources apt for another student who would rather wait some more time for a next turn?.

Code Snippet: - void *studenti(void *i) {}

- This function I used again and again with loop for all the students to acquire the resource and complete the processes.

5. If you have implemented any additional algorithm to support the solution?

- No I have not implemented any additional algorithm to support the solution.

6. Boundary Conditions:-

- Here in this code the boundaries are 3 processes available for students which had to acquire the resources for the completion of their assignment but they each have only 1 resource with them whereas there is 1 teacher with 1 process and unlimited supply of those resources.

7. Output/Test Cases:-

-

```
STUDENT0 has pen

STUDENT1 has paper

STUDENT2 has qpaper

Teacher gives pen,paper

Student2 completed his work


Teacher gives paper,qpaper
```

```
Student0 completed his work


Teacher gives qpaper,pen

Student1 completed his work


Teacher gives pen,paper

Student2 completed his work


Teacher gives paper,qpaper

Student0 completed his work


Teacher gives qpaper,pen

Student1 completed his work
```

8. Have you made minimum 5 revisions of solution on GitHub?

- Yes I had more than 5 revision of solution in GitHub.

Code:-

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<semaphore.h>
#include<pthread.h>
int table_used=1,generated_item[2],generated=0;
char *item[]={"pen","paper","qpaper"};
sem_t table;

void *teacher(void *arg)
{
int i,j,k=0;
int count=0;
```

```
while(1)
{
sleep(1);
sem_wait(&table);
if(count==6) exit(0);
if(table_used==1)
{
i=k;
j=i+1;
if(j==3)
j=0;
k=j;
generated_item[0]=i;
generated_item[1]=j;
printf("Teacher gives %s,%s\n",item[i],item[j]);
generated=1;
table_used=0;
count++;
}
sem_post(&table);

}
}
void *studenti(void *i)
{

while(1)
{
sleep(1);
sem_wait(&table);
if(table_used==0)
{
if(generated && generated_item[0]!=(int)i &&
generated_item[1]!=(int)i)
{
printf("Student%d completed his work\n",(int)i);
printf("\n");

table_used=1;
generated=0;
}
}
sem_post(&table);
```

```
}
}
int main()
{
pthread_t student1,student2,student0,techer;
sem_init(&table,0,1);
printf("STUDENT0 has pen\n");
printf("STUDENT1 has paper\n");
printf("STUDENT2 has qpaper\n");
pthread_create(&techer,0,teacher,0);
pthread_create(&student0,0,studenti,(void*)0);
pthread_create(&student1,0,studenti,(void*)1);
pthread_create(&student2,0,studenti,(void*)2);
while(1);
}
```

Problem 7: Researchers designed one system that classified interactive and non-interactive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as non-interactive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless.

GitHub Link: https://github.com/luciferdevill/OSCA3/blob/master/Q7.c

1. Description: - One of the main jobs of operating system kernels is that they need to be able to execute all of the processes running on your computer efficiently such that high priority processes are completed as quickly as possible, while also ensuring that there is some fairness in how they schedule processes; even if a process is a low priority, it eventually needs to complete execution.

An operating system that allows users to run interactive programs. Pretty much all operating. Systems that are on PCs are interactive OS's Accepting input from a human. Interactive computer systems are programs that allow users to enter data or commands. Most popular programs, such as word processors and spreadsheet applications, are interactive. A non-interactive program is one that, when started, continues without requiring human contact. A compiler is a non-interactive program, as are all batch processing applications. In order to

achieve this, a scheduler looks through the queues for the highest priority process and then executes it for a time quantum (a slice of time), which has been allocated to that queue by the processor. Once the time quantum is up, the process has either completed or it hasn't. If it hasn't, then the process is shunted to the next lower priority queue to wait for its turn again. If the process completed during the initial time quantum, then it gets discarded and the scheduler moves on to the next process in line. Queues are processed in first-in-first-out (FIFO) order.

The time quantum is further divided into smaller amounts of time called work time. A process is allowed to work for this work time, and then the scheduler will begin the loop over. If the process has not finished, and has not become a blocking process, then it will be processed again for the work time. The same process is repeatedly allowed to work until the full quantum time has been reached, or it finishes, at which point, if the process has not finished, it will be sent to the next lower-priority queue, or if already in the lowest priority queue, sent to the back of the line in that queue.

One thing to note is that processes in higher priority queues are allocated less CPU time than processes in a lower priority queue. The logic here is that the scheduler wants to get through as many of the short, high priority processes first, then the long, high priority processes, followed by the short low priority processes, before finally getting around to the long, low priority processes. Oftentimes, these long-running low priority processes only get allocated CPU time when your computer is idle, because during high usage periods, new processes are constantly being added to the highest-priority queue by the scheduler.

2. Algorithm:-

1. When a process starts executing then it first enters the processes you wants basically it asks for the number of processes.
2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.

3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit. In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in FCFS manner.
6. A process in lower priority queue can only execute only when higher priority queues are empty.
7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

3. Complexity: - O(n*log(n))

4. Constraints:-

- Priority Propagation is supported only for per-hop behaviour (PHB)-level classes.
- Priority Propagation works only for the sub-channels within same group (expedite/normal).
- Priority levels for queue-level classes across different sub-channels are treated the same when propagated at the sub-channel level.
- A priority propagated queue does not get priority over queues of other sub-channels that are configured as an expedite queue and CIR is configured on them (that is, priority is set on the sub-channel)
- Priority propagated queue with CIR configured on its sub-channel will compete with HPCT queue traffic.

- **Code Snippet: -** int i, type[20], n, int resptime[20];

- These variables and arrays are used again and again with loop for all the processes to acquire the resources and complete the processes.

5. If you have implemented any additional algorithm to support the solution?

- No I have not implemented any additional algorithm to support the solution.

6. Boundary Conditions:-

- Here in this code the boundaries for the response time given as 1000 and the value should be smaller than it so the processes may execute properly. And the values should not exceeds the arrays limits.

7. Output/Test Cases:-

-

```
Number of process: 5

Enter the data

Response time of P0 (in milliseconds): 30

Response time of P1 (in milliseconds): 10

Response time of P2 (in milliseconds): 50

Response time of P3 (in milliseconds): 60

Response time of P4 (in milliseconds): 40

Process Number  Response Time   Type            Priority

P0              30ms            Interactive Process     High

P1              10ms            Interactive Process     High

P2              50ms            Interactive Process     High

P3              60ms            Interactive Process     High

P4              40ms            Interactive Process     High
```

8. Have you made minimum 5 revisions of solution on GitHub?

- Yes I had more than 5 revision of solution in GitHub.

Code:-

```
#include<stdio.h>
```

```
int main()
{
int i, type[20],n;
int resptime[20];
printf("Number of process: ");
scanf("%d",&n);
printf("Enter the data\n");
for(i=0;i<n;i++)
{
printf("Response time of P%d (in milliseconds): ",i);
scanf("%d",&resptime[i]);
if(resptime[i]<1000)
{
type[i]=1;
}
else
{
type[i]=0;
}
}
printf("Process Number\tResponse Time\tType\t\tPriority");
for(i=0;i<n;i++)
{
printf("\nP%d\t\t%dms\t\t",i,resptime[i]);
```

```
if(type[i]==1)
{
printf("Interactive Process\tHigh");
}
else
{
printf("Non-Interactive Process\tLow");
}
}
}
```